HyperDB: a Novel Key Value Store for Reducing Background Traffic in Heterogeneous SSD Storage

Ruisong Zhou
Huazhong University of Science and Technology
China
ruisongzhou@hust.edu.cn

Yuzhan Zhang
Huazhong University of Science and Technology
China
yuzhanz@hust.edu.cn

Chunhua Li*
Huazhong University of Science and Technology
China
li.chunhua@hust.edu.cn

Ke Zhou
Huazhong University of Science and Technology
China
zhke@hust.edu.cn

Peng Wang
Huawei Technologies Co., Ltd.
Hong Kong
wangpeng423@huawei.com

Gong Zhang
Huawei Technologies Co., Ltd.
Hong Kong
nicholas.zhang@huawei.com

Ji Zhang
Huawei Technologies Co., Ltd.
Switzerland
dr.jizhang@huawei.com

Guangyu Zhang
Huawei Technologies Co., Ltd.
China
zhangguangyu16@huawei.com

ABSTRACT

Log-structured merge tree (LSM-tree) has been widely adopted by modern key-value stores. Deploying LSM-tree across heterogeneous SSD storage which combines the fast but expensive NVMe storage tier with the slow but economical SATA storage tier has emerged as the optimal choice for maximizing cost-effectiveness. However, existing studies typically focus on optimizing the performance of individual storage layers, thereby impeding the full utilization potential of both storage layers. We notice that they tend to over-rely on one storage layer and underutilize the other. In this paper, we present HyperDB, a novel hybrid key-value store designed to enhance the overall performance of both layers via deploying tailored data structures in different media. Especially, HyperDB devises a zone-based data layout for NVMe SSDs to reduce migration overhead, while also implementing a semi-sorted table on the SATA storage layer to minimize merge overhead. Furthermore, we propose a preemptive compaction method at the block-granularity level to further alleviate resource consumption caused by background compaction. Experimental results show that HyperDB achieves 2.25× faster on average throughput and a 60.3% reduction in background task traffic, compared to the standard use of RocksDB in data centers today.

*Corresponding author

CCS CONCEPTS

• **Information systems** → **Hierarchical storage management**; **Storage architectures**.

KEYWORDS

key-value stores, hierarchical storage, data migration, LSM-tree

ACM Reference Format:
Ruisong Zhou, Yuzhan Zhang, Chunhua Li, Ke Zhou, Peng Wang, Gong Zhang, Ji Zhang, and Guangyu Zhang. 2024. HyperDB: a Novel Key Value Store for Reducing Background Traffic in Heterogeneous SSD Storage. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673153

1 INTRODUCTION

Persistent key-value (KV) stores play a crucial role in modern storage systems. By providing APIs for writing, reading, and updating data items, this storage offers an efficient solution to manage, store, and index large volumes of data. The write-friendly log-structured merge tree (LSM-tree) is widely adopted as the underlying storage engine by KV stores, such as RocksDB [7], LevelDB [13] and Cassandra [17], and has been extensively deployed and applied in write-intensive workloads [19].

The emerging commercial NVMe storage has recently experienced widespread adoption, offering lower latency and higher throughput capabilities compared to conventional SATA storage. However, due to the significantly higher cost of NVMe storage in comparison to SATA SSDs, the development of cost-effective multi-tier storage systems has emerged as a preferable choice. This storage system incorporates NVMe storage as a performance tier, alongside conventional SATA storage for a capacity tier. Such systems are currently available among data storage providers and have been deployed for storage-class applications [1–5].

Many studies have shown that simply placing the low-level tier into the high-performance NVMe storage cannot fully utilize the NVMe device's potential. Therefore, several works have attempted to build a more efficient LSM-tree-based embedded architecture in the performance tier such as optimizing WAL [9], memtables [10, 16], index [10, 14, 15, 28] or top levels of LSM-tree [20, 37]. However, these designs either have specific requirements for the performance tier's capacity or struggle to fully leverage the capabilities of the performance tier due to the compaction pressure in the deeper layer. Moreover, some works treat the NVMe storage as a cached architecture to provide extra storage bandwidth [33, 34] and store hot objects [12, 25]. These designs offer the flexibility to adjust the size of storage objects based on available capacity. but mismatches between object size and page size lead to excessive utilization (e.g. read amplification) of the performance layer, especially during migration tasks. According to Cannikin's Law, the performance of KVS storage is limited by the weakest link in the system. As we demonstrate experimentally in Section 2.3, these existing approaches have suboptimal performance for a multi-tiered use case, due to the overutilization or underutilization of NVMe devices in the performance tier, which results in limited potential for performance improvements.

This led us to rethink the architecture of the multi-tier KV store in order to fully leverage the performance of each tier and provide cost-effective services, especially as the capacity tier accounts for a large proportion of the total storage capacity. Our design principle is to minimize the IO occupied and bandwidth utilization by background tasks while maximizing end-to-end read and write performance.

To achieve this goal, we propose a new multi-tier KV store called HyperDB. To fully leverage the random access capabilities of NVMe storage, HyperDB deploys a zone-based layout on the performance tier where each zone stores a part of unsorted objects to support fast random access but ensures the order of objects stored between zones to avoid excessive I/O overhead during migration. HyperDB tries to store frequently read or updated data in NVMe storage, so we use a lightweight object popularity algorithm based on the access interval to estimate the hotness of objects. HyperDB records objects accessed in multiple time windows to determine cold objects and migrate them to the capacity tier. HyperDB also adopts a partitioned, shared-nothing architecture which is independent of zone structures to prevent CPU from becoming a bottleneck.

HyperDB deploys an LSM-tree-based architecture on the capacity layer to make full use of the sequential access capability of SATA SSDs. We redesigned the sorted string table (SSTable) in LSM-tree to the semi-sorted string table (semi-SSTable), which allows appending writes after the file has been persisted, with objects arranged in order within the data blocks, but allows unordered arrangement between data blocks. Exploiting the semi-SSTable data organization, we propose a preemptive compaction method with block granularity to further reduce the resource consumption caused by background compaction and prevent it from becoming a performance bottleneck.

We implement these techniques in HyperDB and compare it favorably against an industry KV store, RocksDB[7] and an academic KV store, PrismDB [25]. We use the industry-standard YCSB benchmarks, with both uniform and skewed key access distributions. Experiment results demonstrate that HyperDB improves the throughput by 2.81× on write-dominated workloads and by 2.27× on read-dominated workloads. Meanwhile, HyperDB also reduces write traffic by 60.3% compared to RocksDB.

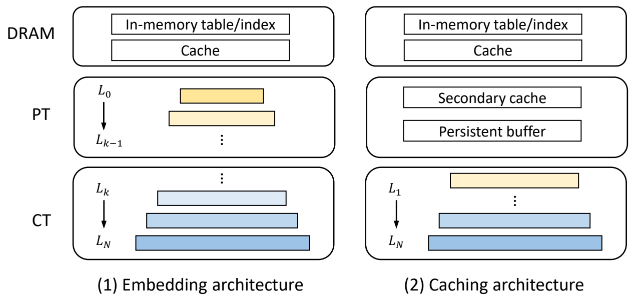


Figure 1: Two architectures for deploying LSM-trees in multi-tier storage, including embedding the top levels in the performance tier (a) and adding an extra cache tier (b).

The rest of the paper is organized as follows. Section 2 introduces the background and motivation for our research. Section 3 describes the design details. Section 4 presents the evaluation results. Section 5 discusses related works and the final section concludes this paper.

2 BACKGROUND AND MOTIVATION

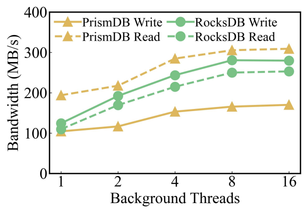
2.1 Log-Structured Merge Trees

The LSM-tree is a multi-level data structure primarily designed for block-based storage devices to handle write-intensive workloads. It utilizes a log structure to convert a batch of random writes into sequential writes, resulting in excellent write performance. Basically, incoming key-value pairs are first appended to a write-ahead log to enable recovery, and them buffered in main memory to construct the MemTable, which is a skip-list and sorted by keys. Once the MemTable is full, it converts to immutable, and a new Memtable is allocated to absorb new writes. In the meanwhile, a background thread is started to flush the Immutable Memtable to persistent storage as an SSTable. Persistent storage is divided into multiple levels of exponentially increasing sizes, from the smallest level L_0 to the largest level L_n. The newly flushed SSTables are first written to L_0 and are compaction for L_0 to deeper levels during the lifespan of LSM-tree.

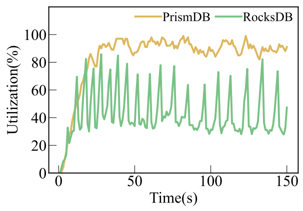
LSM-tree limits the overhead of reads and scans by compaction, keeping each level ordered. The compaction policy controls the order and the number of times a data entry is merged at each level. For example, the level compaction policy selects an SSTable from level L_k and chooses the SSTables from the child level L_{k+1} who has overlapping key ranges. Those SSTables are read into memory to be merged and sorted. The regenerated SSTables are written back to level L_{k+1}. In this process, KV items that are frequently updated will be merged in the shallow level, while less frequently updated KV items will be written to deeper levels through compaction.

2.2 Deployment of LSM-trees in Heterogeneous Storage

With the emergence of high-performance NVMe SSDs, many studies on KV storage have employed hierarchical designs to balance

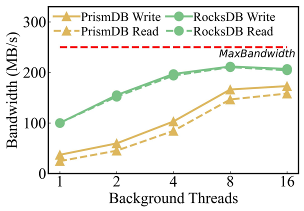


(a) bandwidth utilization

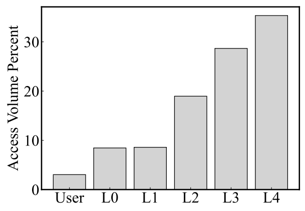


(b) capacity utilization

Figure 2: The utilization of bandwidth (a) and capacity (b) in a multi-tier storage system for RocksDB and PrismDB where NVMe SSD serves as the performance tier and SATA SSD serves as the capacity tier.



(a) Bandwidth used by compaction



(b) Compaction data breakdown

Figure 3: The compaction overhead of the LSM-tree in the capacity tier. The left figure displays the average bandwidth consumed by compaction threads while the right figure provides a detailed breakdown analysis of compaction I/O volume.

the performance and cost by leveraging the performance and capacity tiers cost-effectively. LSM-tree based KV stores have been extended to leverage the hierarchical architecture. These designs can be grouped into two categories, *embedding* and *caching*, as shown in Figure 1. The embedding method puts the recently written data into a performance device in the upper layer, such as the top levels of an LSM tree. For instance, SpanDB [9] and SplitDB [6] store the tree's top levels on NVMe storage. Mutant [38] generates the SSTables in the local SSDs first and then migrates the cold SSTables to the remote SSDs. WALSM [10] also separates the index from the SSTables and places it on NVMe storage.

The caching method copies the recently accessed data from a large-capacity device and temporarily stores newly written data in the upper layer. For example, NoveLSM [16] places the MemTable and immutable Memtable on the NVMe storage to provide fast and durable data writes for key-value put operations. Orthus-KV[34] uses the fast tier as an auxiliary storage tier that provides extra storage bandwidth. And Wu et al. [33] design a database that simply treat the fast storage as an L2 cache.

In summary, both strategies try to leverage high-performance NVMe storage to expand single-level KV storage systems based on LSM-Tree. However, the significant performance gap between high-performance NVMe storage and ordinary NAND storage, as well as the limited capacity in practical applications, have led to KV stores not fully exploiting the feature of multi-tier media storage. We analyze this in detail in the next section.

2.3 Challenges and Motivations

For a multi-tier KV store, the goal is to maximize end-to-end performance by fully utilizing the characteristics of each tier. To explore the challenges in LSM-tree based KV stores, we conduct a preliminary study of LSM-trees within hierarchical architecture. In this section, we undertake a comprehensive analysis of the overall performance of the LSM-tree within hierarchical architecture. We choose RocksDB [7] and PrismDB [25] for our experiments. RocksDB represents the embedding architecture, which combines multiple storage devices through the *db_path*. PrismDB represents the caching architecture deploying a slab-like layout on NVMe storage and migrating objects to the capacity tier through compaction. We generate a dataset containing one billion KV items with 8B key and 128B value size, which is written to databases in a uniformly random order after loading 200GiB datasets. The evaluation environments and other parameters are described in Section 4. The experimental results expose two challenging issues, as we described below.

Resource utilization. We first investigate the device bandwidth utilization of these two designs. We increase the number of background threads and observe the bandwidth utilization of the two tiers. All read operations are triggered by background tasks since the workload comprises only write operations. Figure 2 shows the bandwidth utilization of the NVMe device for the two designs. We observed that the read bandwidth can be up to 1.88× higher than the write bandwidth for PrismDB. Due to the in-place update strategy in NVMe storage, no merge operations are required to maintain data order. Most read operations are triggered by migrations to aggregate KV objects and demote them to the capacity tier. However, modern SSDs operate with a page as the smallest unit of read and write operations. If the object size is smaller than the page size (e.g., 4KB), the unordered data on the disk can result in reading more pages when retrieving consecutive objects from the disk, leading to excessive utilization of the NVMe storage.

For capacity utilization, PrismDB with a cached architecture achieves a utilization rate of over 95% on NVMe storage capacity, while RocksDB's utilization rate ranges between 40%-80%. This discrepancy arises in RocksDB because the size of LSM-tree levels increases exponentially, and a level cannot span across multi-tier storage. When multiple levels are stored in the performance tier, the capacity requirements grow nonlinearly, which limits the flexibility of capacity in multi-tier storage configurations.

Compaction overhead. We then explore the overhead caused by background compaction on these two KV stores in the capacity tier, as the capacity tier has limited bandwidth and is more likely to become a bottleneck. As shown in Figure 3a, with the increase in background threads, RocksDB quickly reaches a bottleneck in bandwidth consumption at the capacity tier, utilizing up to 91.3% of the device's bandwidth. PrismDB has a slow compaction speed for a single thread, but there is also significant overhead when using multiple threads. Due to the more efficient use of space in the performance tier, PrismDB reduces the number of levels stored in the capacity tier, resulting in lower write bandwidth compared to RocksDB. We then break down the compaction overhead for each level in Figure 3b, in a RocksDB key-value store with five levels, 38% of the compaction overhead is attributed to L4, which shows that most of the compaction overhead comes from deeper layers. This is because when compacting at higher levels, there is a greater likelihood of overlapping data ranges among the SSTables being merged, leading to more data being rewritten and moved during the compaction process.

Summary. In this section, we investigate and analyze the storage utilization of two multi-tier KV store architectures. As we can see, in current storage solutions, there are still limitations in terms of scalability and flexibility, making it challenging to fully leverage the potential of each tier in a multi-tier storage environment. This motivates us to architect a new data layout and compaction mechanism that is optimized for multi-tiered storage.

3 DESIGN AND IMPLEMENTATION

In this section, we present an efficient solution called *HyperDB* to address the aforementioned limitations. In our design, we utilize NVMe SSDs as the performance tier (referred to as NVMe storage) and SATA SSDs as the capacity tier (referred to as SATA storage).

3.1 Design Overview

Figure 4 depicts the architecture of HyperDB. HyperDB uses a hybrid data layout optimized for both tiers. Inspired by KVell [18], HyperDB employs a shared-nothing architecture in the NVMe storage by dividing the entire space into partitions. Each Partition manages its own metadata and lock, while the partitions collectively share the bandwidth and capacity resources of the hardware.

The NVMe storage buffers all write requests and serves as a secondary cache after DRAM for hot objects, it also stores a backup of the index and metadata of objects within the LSM-tree in the SATA storage for low-cost index lookup. Since the metadata block and index block only occupy a small portion compared to data blocks, storing them entirely in NVMe storage would not significantly consume much space.

HyperDB uses different data structures for NVMe and SATA storage. In NVMe storage, we implemented a zone-based structure; for SATA storage, we redesigned the file format in the LSM-tree to a semi-SSTable and optimized the merging operations during data compaction. Additionally, HyperDB developed a lightweight object hotness tracker that can distinguish and place hot and cold objects in different zones. Furthermore, a new compaction algorithm was introduced for the semi-SSTable, significantly reducing data duplication during compaction across multiple levels. We then describe these technologies in detail.

3.2 Data Structure

Zone-based structure. As shown in Figure 4, the zone group is the basic component for persistence in the NVMe storage. Each zone group has several zones, where a zone stores a collection of items with adjacent key ranges, and the key range between zones is ordered and non-overlapping. Each zone has a series of slots, where items are allocated to the corresponding slot based on their value size. The zone mapper maps the pages from slot files for each zone. When a new object is written to the zone, a new page is applied in the corresponding slot and added to the zone mapper. HyperDB

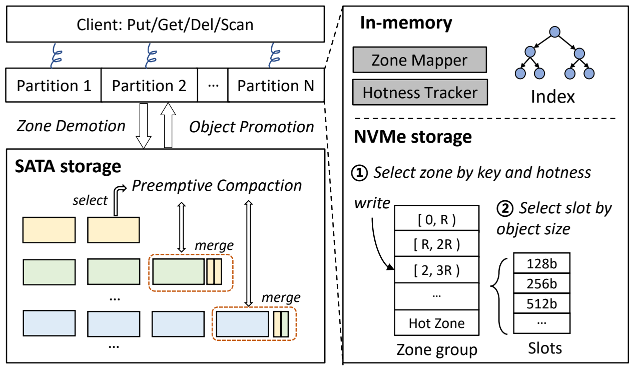


Figure 4: Architectural Overview of HyperDB

accesses slots at block granularity, which is the page size on our machines. HyperDB prefixes objects in the slot with a timestamp, the key size, and the value size. For objects smaller than the page size, updates are performed in place. For larger or resized items, updates are written to a new block, with a tombstone written at the original item's location.

Each zone only stores objects with a limited key range. We set the zone capacity as the migration batch size and estimate the key range size of the zone. HyperDB periodically rebuilds the zone size based on the workload and updates the representation range when creating a new zone. For a zone group with K unique slot files, each slot file k has written N_k objects and the file size is F_k. Thus, we can estimate the average object size of the partition as follows, where B represents the preset migration batch size:

$$O_k = \frac{\sum_{i=1}^{K} F_k}{\sum_{i=1}^{K} N_k} \tag{1}$$

Then we can calculate the key range representation of zone R_k as:

$$R_k = \frac{B}{O_k} \tag{2}$$

Zone structure limits the range of data written to the page, which enhances the efficiency of the sequence scan. Therefore, it is straightforward to select all pages from a zone and construct a migration batch with a limited key range. Although the data pages within a zone are discontinuous in the file, this does not become a bottleneck due to the excellent random read capabilities of NVMe storage.

HyperDB does not migrate frequently accessed data in the short term. Consequently, we have established a hot zone to accommodate data identified as hot by the tracker. The hot zone imposes no restrictions on key ranges. When objects remain unaccessed for a period and are reclassified as cold, they are transferred to other zones for migration.

Optimizing LSM-tree in SATA storage. The data layout of SATA storage is optimized for sequential writes using an LSM-tree. As shown in Figure 5, each file in LSM-tree is a semi-SSTable, which supports append operations post-persistence. Each semi-SSTable consists of three parts: the data block, metadata blocks, and index blocks. The data block stores the sorted KV entries and the metadata

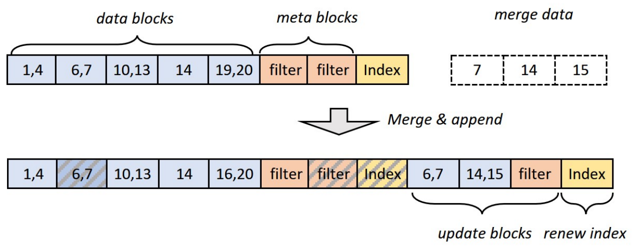
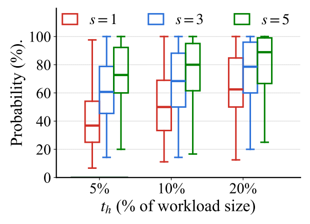
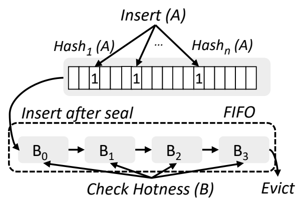


Figure 5: An example of a semi-SSTable and block-level merge operation is presented. Three new objects are merged into the original table, the blocks that need to be rewritten are considered dirty blocks. Objects from the dirty blocks and objects that need to be merged form a new block and the index is rebuilt.



(a) Inferring next access interval



(b) The tracker overview

Figure 6: The strong correlation between historical access intervals and the next visit time (a). And the tracker design (b) which employs a cascading discriminator to track access.

blocks contain bloom filters that provide fast lookup for entries. The index blocks not only store the offsets, key ranges, and validity of each data block but also use prefix compression to store all valid keys within the entire SSTable. During merges, any data blocks needing rewrites are labeled as dirty, whereas unchanged blocks are marked as clean and kept for future use. The system then sorts and merges these objects with those pending integrations, appending new blocks to the file's end while generating fresh metadata and index blocks.

HyperDB imposes restrictions on the key range of individual file storage objects to reduce the overlapping of file key ranges during the compression process at deeper levels. For the largest level L_n, we divide the key space into segments uniformly, with each file storing objects corresponding to on or multiple segments until the file size exceeds the threshold. Level $n-1$ (the second largest level) is also partitioned into files such that the key range of each file overlaps with T contiguous files in L_n, where T is the LSM-tree size ratio. Smaller layers are configured in the same manner. We set the first level as L_1, since the NVMe storage can be seen as L_0. Objects migrating from NVMe to SATA storage will be merged into the appropriate files in L_1 based on the representational range of each file at that level. This can also avoid reducing compression efficiency due to the large overlap of L_0 files [39].

3.3 Track the Hotness of Objects

When the NVMe storage's used capacity hits a high watermark, the partition triggers a background compaction job to demote colder objects to the SATA storage to free up space. Since the access patterns in real-world workloads are often highly skewed, it's essential to identify hot data quickly and accurately at a low cost. There is a large body of work on how to track and estimate object popularity [29, 30, 35]. However, most of them require relatively high computing and storage resources. Since the KV pairs are often small, we need to limit the metadata for tracking the characteristics of objects. We now discuss in more detail how HyperDB tracks the popularity of hot objects and designates them as hot or not.

Estimate the hotness with access interval. HyperDB tracks the hot objects by access interval, which has been shown to be effective in identifying objects with a limited lifespan[24, 31]. Based on the historical access interval of an object, one can infer its future access interval, thereby determining the hotness of the data. To demonstrate the correlation of the access interval, we construct and replay a workload where 80% of the accesses are concentrated on 20% of the objects, and depict the relevant probability $P(t_n < t \mid t_h < t)$ of the next access interval t_n and the historical access intervals t_h for s times. We set t_n as the same as t_s and Figure 6a shows the boxplots of the conditional probabilities over all access objects for different t_n and s. In general, the conditional probability remains high for most of the objects. For example, for t_n being 20% of the workload size, the medians of the conditional probability are 62.5% for $s = 1$ and 88.9% for $s = 5$, and the 75th percentile probability is 84.8% for $s = 1$ and 99.0% for $s = 5$. The conditional probability increases with the enlargement of s. Therefore, we identify the objects with continuous access intervals less than the threshold condition as hot, we set the threshold as the number of objects that NVMe storage can store, and keep the hot objects retained within NVMe storage.

Track the hotness with cascading discriminator. To minimize the overhead of tracking the hotness of objects, we do not need to know the specific value of the object access interval, but only whether it meets the threshold condition. Figure 6b illustrates the tracker of HyperDB. We deploy a cascading discriminator for each partition to record and detect the objects with low memory and calculation overhead. The cascading discriminator consists of several standard bloom filters, each bloom filter can count the inserted data by assessing whether any bits have been modified. At a lower false positive rate, the size of a bloom filter can represent an access window, and finding an object in the bloom filter can represent the data access interval being less than this window. To mitigate errors caused by single-window judgments, we utilize a cascading discriminator to employ multiple windows for joint evaluation.

We open a bloom filter during the tracker initialization. Each read or update operation from clients requires the tracker to insert the object into the opening bloom filter. When the bloom filter reaches its capacity, the tracker sets it as sealed and adds it to the cascading discriminator. The tracker restricts the number of elements in the cascading discriminator and evicts the oldest bloom filter in a FIFO manner. After the insertion, the tracker verifies the existence of the object in all sealed bloom filters and identifies it as hot data only when it is present in a continuous series of bloom filters.

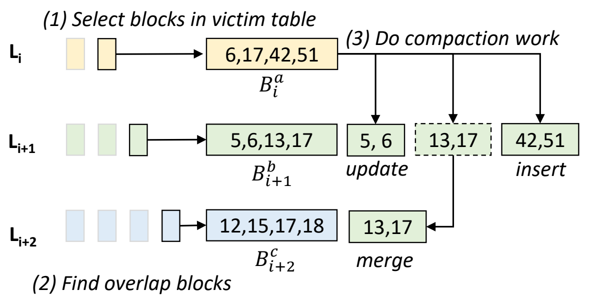


Figure 7: An example of preemptive block compaction. For a victim table in L_i, we update object (6), insert objects (42,51) into L_{i+1}, and preemptively merge object (17) with object (13) from L_{i+1} into L_{i+2}.

3.4 Preemptive Block Compaction

As mentioned in §2.3, the LSM-tree compaction often leads to many repeated data writes, resulting in higher write amplification and becoming a bottleneck for the entire system. By exploiting the semi-SSTable data organization, we propose a preemptive compaction method with block granularity for SATA storage to further reduce the resource consumed by background compaction and prevent it from becoming a performance bottleneck.

Compaction process. As illustrated in Figure 7, suppose the size of an LSM tree level L_i exceeds the predefined limit, the preemptive block compaction selects a victim semi-SSTable and identifies those tables in level L_{i+1} whose key ranges overlap with those of blocks in the victim semi-SSTable. For blocks B_{i+1}^a in L_{i+1} with overlapping keys, we search for blocks in the subsequent level L_{i+2} with overlapping keys with B_{i+1}^a, and this process is recursively repeated until reaching L_{i+k}.

In this way, files from the k levels that have overlapping keys with blocks in the previous level are selected as input files. Then we commence the compaction work. First, multiple worker threads read the keys of the selected tables and find overlapping blocks in a top-down manner. For fast calculation, the index blocks store all keys from the data blocks and the workers only read the index blocks to retrieve keys for each table. Then we read all blocks in L_i and corresponding overlapping blocks in other levels and try to construct new blocks with these objects in three ways. We construct objects with continuous keys that do not overlap with existing blocks in L_{i+1} as new blocks and insert them into L_{i+1}. Similarly, If these consecutive keys are updated in L_{i+1}, the object will be updated for the corresponding blocks and form new blocks. A special case is when there are also repeated keys in the next level, e.g., L_{i+2}, we merge these overlapping objects into a deeper level.

We also need to determine the compaction method and perform compaction for each table. Since always doing block compaction may cause a large number of dirty blocks and result in more space overhead, we establish a threshold T_{clean} (e.g., 50%) and consider full compaction if the ratio of dirty blocks in a table exceeds T_{clean}. Regular full compaction can enhance the organization of data within the table, leading to improved performance during sequential reads.

Victim tables selection. Traditional compaction typically selects victim tables in a round-robin policy or selects the tables with the least overlap with the target level to balance write amplification and space amplification. In our design, preemptive compression to a deeper level can reduce write amplification, but it may accumulate more dirty blocks, leading to a larger space overhead. Thus, a good trade-off is necessary from these two perspectives. Therefore, we select victim tables based on the situation of space overhead. When the space overhead exceeds predefined values (e.g. 1.5× space amplification), we select the victim table with the dirtiest blocks to free more space. Otherwise, we select the table with the highest overlap scores, which means merging blocks to the highest level as much as possible.

We calculate the overlap score as described in Algorithm 1. We define the *blockmeta* variable as the key range of blocks from one level's files that overlap with selected blocks on the previous level. For a candidate table T in level L_k, we identify those tables in level L_{k+1} whose key ranges of blocks overlap with the blocks of the victim semi-SSTable. Furthermore, the search for overlapping blocks in level L_{k+1} is based on the actual key range of blocks with overlapping key ranges in level L_k. The level score is the number of blocks with overlapping key ranges in these files. We then sum the level scores to calculate the overlap score of the selected table. Calculating the scores for each table in selected levels is expensive, as there may be many tables. Therefore, we use a power-of-k choices method[23] to select a subset of tables as candidates, which can efficiently reduce the computing overhead for large databases. We empirically use $k = 8$, as it provides a good trade-off with acceptable compute overhead.

Algorithm 1: Identify a suitable victim table by evaluating the overlap score.

```
Input: Compaction level $i$, compaction depth $k$, Candidate
       tables $C$,
Output: The score for each candidate table ScoreMap
Initialize BlockMeta[$i + k$], ScoreMap;
for Table $T \leftarrow C$ do
    BlockMeta[$i$] $\leftarrow$ $T.blocks$ ;
    for $n \leftarrow 1$ to $k$ do
        for Block $b \leftarrow$ BlockMeta[$i + n - 1$] do
            $blocks \leftarrow$ FindOverlapBlocks($b, L_{i+n}.tables$);
            BlockMeta[$i + n$].$merge(blocks)$;
        end
        $score \leftarrow score +$ BlockMeta[$i + n$].$size()$;
    end
    ScoreMap.$insert(T, score)$;
end
```

3.5 Migration Cross Tiers

HyperDB monitors the usage capacity of NVMe storage and performs migration tasks. When it reaches a high watermark, HyperDB triggers background demotion migration jobs to demote cold objects to SATA storage until it frees up enough space to absorb new incoming writes. When a read operation targets hot objects in the SATA storage, it promotes these objects and inserts them into the hot zones.

Zone demotions. Although NVMe storage offers outstanding random read capabilities, the disproportionate sizes of objects and pages can lead to significant resource wastage. HyperDB strives to select the fewest pages to migrate cold objects with the narrowest key ranges. As HyperDB has partitioned objects into zones and ensured that key ranges of objects between zones in each slab file do not overlap, it selects a zone to migrate once a time. A modified cost-benefit approach [25, 27] is adopted to assess the benefits and costs associated with reading and migrating zones within a key range.

Specifically, we define the demotion benefit for a key range with t objects as the freed-up capacity in NVMe storage, which is calculated by the size of KV pairs selected as $\sum_{i=1}^{t} obj.size$. The cost is the number of read I/Os from NVMe storage for accessing these objects. Thus, we can calculate the metric of our migration schedule as the ratio of benefit to cost as $\frac{\text{benefit}}{\text{cost}} = \frac{\sum_{i=1}^{t} obj.size}{\text{read I/Os}}$.

Since the key range has already been divided into zones, when NVMe storage meets the migration water level and requires demotion, HyperDB calculates the metric for each zone and selects the key range with the highest score for migration. Each zone maintains a counter to track the number of read I/O accesses within an interval. The counter for each zone is reset to zero after each migration.

Object promotions. During the migration process, objects can be both promoted and demoted. In read-heavy workloads, NVMe storage only gradually fills up, and most operations are read-oriented and hit the performance tier. In this scenario, it's also necessary to promote objects from low-speed devices to NVMe storage to enable fast reads for popular objects. Therefore, when read operations access hot objects in SATA storage and the object is identified as hot in the tracker, HyperDB promotes the object to NVMe storage.

Hot objects will initially enter the object cache in memory and asynchronously flush to the corresponding hot zone in NVMe storage upon eviction, with their positions marked in the index and labeled for *promotion*. When eviction is necessary for the hot zone, objects no longer recognized as popular and bearing the *promotion* labels will be eliminated directly, without undergoing additional relocation.

3.6 HyperDB Implementation

In this section, we describe the implementation of our design. HyperDB is written in C++ and built upon LevelDB [13]. We redesigned the data layout of NVMe storage from KVell [18]. Additionally, the semi-SSTable structure is inspired by BlockDB [32]. By default, we use a compaction thread for each partition, which runs in the background and is intermittently triggered to free space. Promotions are triggered when hot objects read from the SATA storage, and we add it to the NVMe storage asynchronously.

Index. We use an in-memory B-tree index to locate the objects stored in the NVMe storage. Each index entry stores the key and its address in NVMe storage. Items are indexed by the prefix of their keys to preserve the order for range scans. The objects in the SATA storage

Partition and zone settings. To maximize the performance of NVMe storage, we empirically use 8 partitions for each NVMe device, since this configuration can realize the full potential of the devices without wasting CPU resources. The zone mapper keeps track of the pages in each zone, along with the zone's size and read I/O counts for each period. The size of zones limits the range of keys for objects in a page. More zones keep the data more ordered between pages but cause more memory overhead. In our design, we set the zone size to be the same as the semi-SST file size, which allows for better migration efficiency.

Cascading discriminator. The tracker for each partition has a cascading discriminator. In each bloom filter, we set 10 bits for an object to keep the false positive probability less than 1%. We set the capacity of a bloom filter based on the estimated number of objects that the partition can store. Moreover, the cascading discriminator can maintain up to four bloom filters. We classify an object as hot when it is recognized in at least three Bloom filters.

4 EVALUATION

4.1 Evaluation Setup

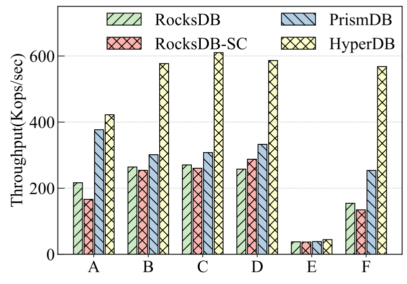
Settings. We performed our experiments on a machine with a 16-core Intel Xeon Silver 4314 CPU and 64 GB of memory on Linux(Ubuntu 22.04). In our experiment, we used a 960 GB Samsung PM9A3 and a 960 GB Intel D3-S4610 as the NVMe storage and the SATA storage. We employed a CPU cgroup of 8 cores in our experiments to fully utilize the NVMe SSD's bandwidth. We set the partition number to 8 with each partition configuring a background thread for migration. All partitions shared a total of 64MB LRU cache in DRAM at the page granularity for admission and eviction. The LSM-tree based capacity tier was configured with 1 background compaction thread for each partition.

Baselines. We compared HyperDB against three baselines: (1) RocksDB v7.10 [7], an LSM KV store developed by Facebook and heavily used in industry. (2) RocksDB with NVMe storage used as a secondary read cache (labeled RocksDB-SC). (3) PrismDB [25], which is a hybrid KV store with efficient compactions between storage tiers. For these baselines, we also set eight clients and eight background threads. We used default RocksDB settings, with a 64MB SSTable size and asynchronous WAL writes.

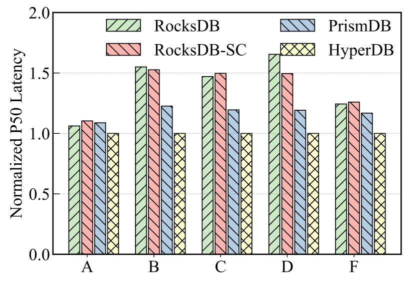
Datasets. We used the YCSB [11] cloud benchmark to generate workloads with varying search/update ratios and Zipfian ratios. The default key and value sizes were 8 bytes and 128 bytes. Before each experiment, we first loaded the KV store with 100 GB of randomly generated KV pairs. Then we performed 100M corresponding search/update requests based on the workload type. The default scan length of range queries was 50 KV pairs, which is a widely used configuration in performance evaluations.

4.2 Overall performance

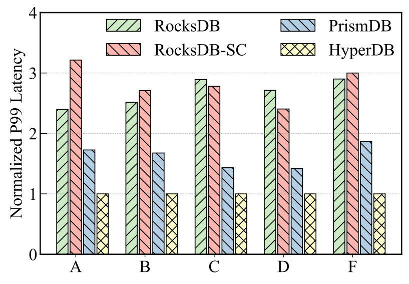
YCSB evaluation. Figure 8 compares the throughput, median, and 99th percentile latencies between different YCSB workloads. We observe that the throughput of RocksDB-SC performs better than RocksDB only in the YCSB-D workload with a 12% improvement and worse in the other workloads, this is because only the YCSB-D



(a)

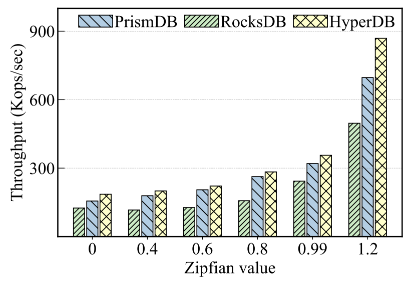


(b)

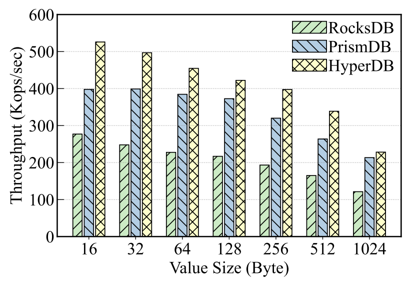


(c)

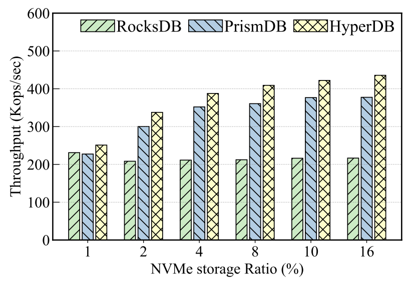
Figure 8: YCSB benchmark results for throughput (a), normalized median latency (b), and normalized P99 latency (c).



(a) Impact of the workload skewness



(b) Impact of the workload value size



(c) Impact of the NVMe storage ratio

Figure 9: The impact of varying workloads (a, b) and heterogeneous storage ratios (c) on performance.

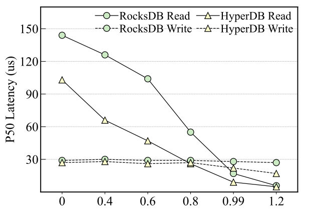
workload can fully benefit from the secondary cache, as it is characterized by reading the latest objects. For the other workloads, the secondary cache suffers from cold starts, and allocating all levels in the capacity tier will cause higher extra write volume, resulting in performance less than that of RocksDB. PrismDB outperforms RocksDB in YCSB-A with 1.74× improvement in throughput due to its efficient utilization of storage space. HyperDB shows the best performance in throughput among all methods, with 2.18-2.27× for read-intensive workloads(YCSB-B, C and D). In these workloads, frequently accessed objects are written to or promoted to the hot zones with the majority of pages being cached. HyperDB also offers an average 2.81× throughput improvement for write-intensive workloads (YCSB-A and YCSB-F). In these workloads, HyperDB requires fewer bandwidth resources during migration compared to PrismDB, thus retaining more bandwidth for foreground tasks to prevent slowdowns, resulting in better performance improvements.

For the scan workload (YCSB-E), HyperDB shows no performance improvement compared to the baselines. This is for two reasons. First, RocksDB strictly sorts keys at each level, which helps reduce data block reads during scan operations. Additionally, it utilizes a prefetcher that proactively fetches blocks, significantly enhancing its performance for predictable scan patterns. In contrast, HyperDB's different data layouts for performance and capacity tiers complicate the prefetcher design. In our implementation, scans are now conducted through sequential point queries. We will optimize this as part of our future work.

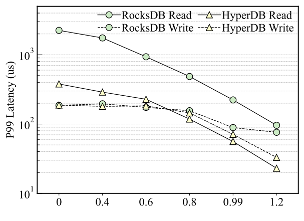
HyperDB also demonstrates good performance in latency metrics, especially for tail latency. It achieves a 5.8%-39% reduction in median latency and a 58.2-65.5% reduction in P99 latency compared to RocksDB. For write-intensive workloads, HyperDB collects and flushes a batch of objects with a zone per migration task, which reduces page reads by 72% compared to PrismDB.

Impact of data skewness. Subsequently, we assess the influence of different workloads and NVMe storage ratios on HyperDB. We first evaluate how data skewness impacts HyperDB. Figure 9a presents the throughput result of a key distribution sweep using YCSB-A. HyperDB achieves 1.48-1.80× improvements compared to RocksDB and 1.08-1.25× improvements compared to PrismDB. For highly-skewed workloads, HyperDB delivers better performance because of the hotness tracking, with more read and write accesses hitting the NVMe storage. For uniform workloads, the throughput shows a modest enhancement and would not be inferior to RocksDB.

We also break down the latency of different workload skewness. As shown in Figure 10, compared to RocksDB, HyperDB exhibits significantly lower read latency, whether considering median or tail latency. Due to the effective promotion of hot objects and the full utilization of NVMe storage capacity enabling the caching of more objects, HyperDB can achieve up to 54.8% median latency reduction and up to 83.4% P99 tail latency reduction for read operations. However, HyperDB finds it challenging to achieve substantial enhancements in write latency. This is attributed to RocksDB's well-optimized approach to write latency, leveraging group commit writes to the WAL to reduce average persistence time, with the time taken to write to the memtable being negligible.



(a) Medium latency



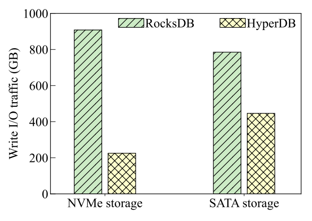
(b) P95 latency

Figure 10: Read/write latency breakdown with different workload skewness.

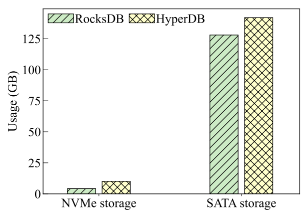
Impact of value size. We then evaluate the impact of value size on the performance of the KV store. Figure 9b shows the throughput performance of different value sizes. We observe that RocksDB and HyperDB exhibit throughput improvements of 27.1% and 24.5%, respectively, when the value size is 16 bytes compared to 128 bytes, whereas PrismDB only shows a 6.7% enhancement. This is because with a smaller value size, PrismDB needs to read more keys during a migration, consequently accessing more pages. Compared to a value size of 128 bytes, when the value size is 16 bytes, PrismDB reads 4.8× more keys during a migration, resulting in a 2.97× increase in migration time per operation. In contrast, HyperDB's migration time per operation only increases by 8%. As the workload value size increases and approaches the page size, the extent of performance improvement in HyperDB decreases, gradually approximating that of PrismDB. At this point, HyperDB exhibits a 1.88-2.05× throughput increase compared to RocksDB.

Impact of NVMe storage ratio. We evaluate the impact of the NVMe storage ratio. In this experiment, We perform a 100GB size request for load and set the NVMe storage capacity based on the ratio (e.g. 1GB for a 1% ratio), and perform 100M YCSB-A requests for evaluation. Figure 9c illustrates the result. We find that RocksDB does not exhibit significant performance improvements with the increase in NVMe storage capacity. In contrast, PrismDB and HyperDB have shown respective enhancements of 1.66× and 1.73× at a 16% ratio compared to a 1% capacity.

Background traffic. We have identified compaction as the primary factor limiting write performance. This section analyzes HyperDB's optimization for storage write volume. Lower background write volumes can prevent the premature exhaustion of dense flash lifespan. We loaded 100GB of data and ran a 100M YCSB-A workload with a uniform distribution and a 1KB value size, then recorded the write volume and space usage for each tier. As shown in Figure 11, HyperDB exhibits the lowest write volume compared to all baselines. Relative to RocksDB, it decreases NVMe storage writes by 75.2% and SATA storage writes by 43.1%, culminating in an overall reduction of 60.3%. HyperDB also demonstrates better utilization of NVMe storage space, although it results in a 10.9% increase in space consumption on SATA storage. This increase is attributable to the use of semi-SSTables that allow for the presence of stale data within levels and engage in full compactions lazily.



(a)



(b)

Figure 11: Total write I/O traffic (a) and space usage (b) under a uniform distribution workload.

5 RELATED WORK

Many works attempt to construct a highly efficient KV store in multi-tier storage devices. We discuss some related works pertinent to our design.

Explore the potential of NVMe storage. Previous research has investigated the eutilization of high-speed NVMe storage to enhance performance[9, 14–16, 25]. For einstance, SLM-DB [15] and LightKV [14] store values on SSDs while placing all keys on NVMe storage to speed up search performance. NovelLSM [16] extends MemTable into NVMe storage to handle small updates efficiently. WaLSM [10] deploys an ART-based index to reduce duplicate data in NVMe storage. SpanDB [9] places the top levels in NVMe storage and uses SPDK suite to reduce latency. PrismDB [25] employs a clock-based method to identify the hotness of objects and considers both benefits and costs during compaction. Although these studies extensively utilize the performance capabilities of NVMe storage, they overlook the potential bottlenecks caused by the underlying storage tier.

Reduce background tasks overhead. We further discuss several works that aim to mitigate write amplification resulting from background compaction [8, 18, 21, 22, 26, 36]. WiscKey [22] and HashKV [8] separate keys and values and only maintain keys in the LSM-tree to reduce write amplification. DiffKV [21] further develops fine-grained KV separation algorithms, acknowledging that applying KV separation to small KV pairs is not always beneficial. PebblesDB [26] allows SSTables stored at the same level to have overlapping key ranges.LWC-tree [36] proposes a lightweight compaction method that appends data to SSTables in the next level and only merges the metadata. KVell [18] implements an in-place update design for high-performance KV stores with novolatile memory, which abandons the log-structured architecture and eliminates write amplification. However, they are optimized solely for single-tier storage and pose challenges in deployment into multi-tier storage systems.

6 CONCLUSION

In this paper, we present HyperDB, a novel key-value store for heterogeneous SSD storage. We introduce three technologies to reduce background traffic. To minimize read overhead caused by migration tasks, we deploy a zone-based layout in NVMe storage, ensuring that objects within a key range are placed in the same zone. We also design a lightweight hotness tracker to detect the popularity of objects and guide migration tasks to reduce traffic effectively. We implement a semi-sorted table structure on the SATA storage and propose a block granularity preemptive compaction method to further reduce the resource consumption caused by background compaction. The experimental results demonstrate the superiority of our proposal.

ACKNOWLEDGMENTS

We would like to express our profound gratitude to the reviewers for their insightful comments and suggestions. This work was supported by the National Key Research and Development Program of China (Grant No.2023YFB4502701), and partially supported by the National Natural Science Foundation of China (No.62232007, No.61821003).

REFERENCES

[1] 2024. Amazon Hybrid Cloud Storage. https://aws.amazon.com/cn/products/storage/hybrid-cloud-storage/.
[2] 2024. Dell Unity XT Hybrid Unified Storage. https://www.dell.com/en-us/dt/storage/unity.htm.
[3] 2024. Google Cloud Storage. https://cloud.google.com/.
[4] 2024. Huawei OceanStor Hybrid Flash Storage. https://e.huawei.com/en/products/storage/hybrid-flash-storage.
[5] 2024. Microsoft Azure. https://azure.microsoft.com/en-us/products/storage/disks/.
[6] M. Cai, X. Jiang, J. Shen, and B. Ye. 2024. SplitDB: Closing the Performance Gap for LSM-Tree-Based Key-Value Stores. *IEEE Trans. Comput.* 73, 01 (jan 2024), 206–220. https://doi.org/10.1109/TC.2023.3326982
[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and DavidHung-Chang Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. *File and Storage Technologies,File and Storage Technologies* (Jan 2020).
[8] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1007–1019. https://www.usenix.org/conference/atc18/presentation/chan
[9] Hao Chen, Chaoyi Ruan, Li Cheng, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. *File and Storage Technologies,File and Storage Technologies* (Jan 2021).
[10] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-Aware Log-Structured Merge Key-Value Store for NVM-SSD Hybrid Storage. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde55515.2023.00171
[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152
[12] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. *Networked Systems Design and Implementation,Networked Systems Design and Implementation* (Jan 2019).
[13] Google. [n. d.]. LevelDB. https://github.com/google/leveldb.
[14] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. LightKV: A cross media key value store with persistent memory to cut long tail latency. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST'20)*.
[15] Olzhas Kaiyrakhmet, SongYi Lee, Beomseok Nam, SamH. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. *File and Storage Technologies,File and Storage Technologies* (Feb 2019).
[16] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, AndreaC. Arpaci-Dusseau, and RemziH. Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. *USENIX Annual Technical Conference,USENIX Annual Technical Conference* (Jul 2018).
[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.
[18] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. https://doi.org/10.1145/3341301.3359628
[19] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. 2023. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. *ACM Transactions on Storage* 19, 2 (2023), 1–32.
[20] Jinhong Li, Qiuping Wang, and Patrick P. C. Lee. 2022. Efficient LSM-Tree Key-Value Data Management on Hybrid SSD/HDD Zoned Storage. arXiv:2205.11753 [cs.PF]
[21] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 673–687. https://www.usenix.org/conference/atc21/presentation/li-yongkun
[22] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu
[23] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* (Jan 2001), 1094–1104. https://doi.org/10.1109/71.963420
[24] Seonggyun Oh, Jeeyun Kim, Soyoung Han, Jaeho Kim, Sungjin Lee, and Sam H. Noh. 2024. MIDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 259–275. https://www.usenix.org/conference/fast24/presentation/oh
[25] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 179–193. https://doi.org/10.1145/3582016.3582052
[26] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.
[27] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* (Feb 1992), 26–52. https://doi.org/10.1145/146941.146943
[28] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 588–602. https://doi.org/10.1145/3575693.3575722
[29] Zhenyu Song, Daniel Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. *Networked Systems Design and Implementation,Networked Systems Design and Implementation* (Jan 2020).
[30] Yujuan Tan, Baiping Wang, Zhichao Yan, Witawas Srisa-an, Xianzhang Chen, and Duo Liu. 2020. APMigration: Improving Performance of Hybrid Memory Performance via An Adaptive Page Migration Method. *IEEE Transactions on Parallel and Distributed Systems* 31, 2 (2020), 266–278. https://doi.org/10.1109/TPDS.2019.2933521
[31] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 429–444. https://www.usenix.org/conference/fast22/presentation/wang
[32] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing Write Amplification of LSM-Tree with Block-Grained Compaction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3119–3131. https://doi.org/10.1109/ICDE53745.2022.00279
[33] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. 2019. Exploiting Intel Optane SSD for Microsoft SQL Server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*. New York, NY, USA. https://doi.org/10.1145/3329785.3329916
[34] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, AndreaC. Arpaci-Dusseau, and RemziH. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. *File and Storage Technologies,File and Storage Technologies* (Jan 2021).
[35] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 115–134. https://www.usenix.org/conference/fast23/presentation/yang-juncheng
[36] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Transactions on Storage* (Nov 2017), 1–28. https://doi.org/10.1145/3139922
[37] Ting Yao, Yiwen Zhang, Jiguang Wan, Chen Qiu, Tang Liu, Hong Jiang, Conghua Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. *USENIX Annual Technical Conference,USENIX Annual Technical Conference* (Jan 2020).

[38] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 162–173. https://doi.org/10.1145/3267809.3267846

[39] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 65–80. https://www.usenix.org/conference/fast23/presentation/yu